



Tema 5. Aplicaciones WPF

Desarrollo de Interfaces
DAM – IES Doctor Balmis



Javier Catalá

INDICE

1. Ventanas
2. Diálogos
3. Comandos
4. Menús
5. Barra de herramientas y barra de estado
6. Configuración de aplicaciones
7. Tooltips
8. Mejora de la productividad

1. Ventanas

Propiedades de la ventana

Propiedad	Descripción
WindowState	Controla el estado de la ventana (Normal, Maximizada o Minimizada)
WindowStyle	Determina el tipo de borde de la ventana y la barra de título
ShowInTaskBar	Decide si la ventana aparece o no en la barra de tareas de Windows
ResizeMode	Controla como el usuario puede redimensionar la ventana
WindowStartupLocation	Determina como se establece la posición inicial de la ventana
Top/Left	Coordenadas de la posición de la ventana
SizeToContent	Decide si la ventana se debe ajustar a su contenido
TopMost	Determina si la ventana debe estar siempre por encima del resto

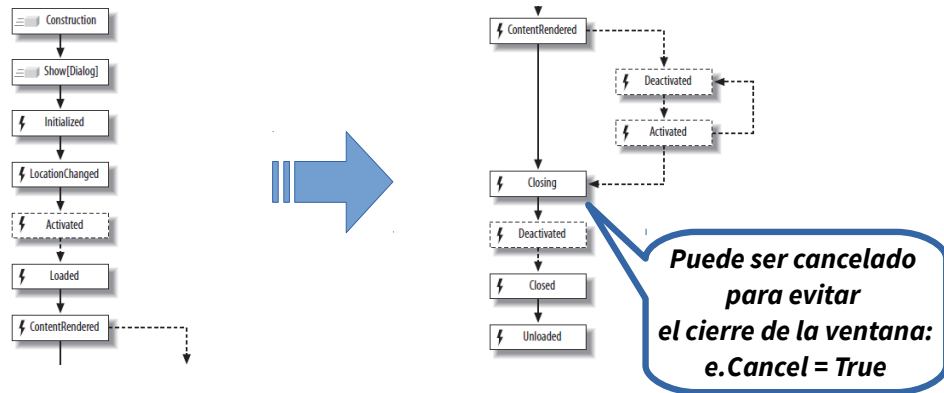
Ya hemos utilizado otras propiedades importantes de la ventana como *Icon*, *Title*, *Width/Height*, *MaxWidth/MaxHeight* y *MinWidth/MinHeight*.

A lo largo de los temas anteriores hemos ido utilizando algunas propiedades de la ventana como *Title*, *Icon* o las relacionadas con el tamaño (*Width*, *Height* y las que permiten definir un alto y ancho mínimo y máximo).

La ventana cuenta con muchas otras propiedades, algunas de las cuales se recogen en esta diapositiva. Por ejemplo, *ResizeMode* permite decidir si el usuario podrá o no redimensionar la ventana y de que forma (con los botones de minimizar y maximizar, con el borde de la ventana, con un control en la esquina inferior derecha, ...). Con *WindowStartupLocation* podemos determinar la posición inicial de la ventana (centrada en la pantalla, centrada en su ventana padre, establecer manualmente la posición,...).

1. Ventanas

Eventos de la ventana



En esta diapositiva vemos la secuencia de eventos que se desencadenan en la vida de una ventana.

Hay que destacar el evento *Closing*, que se produce cuando el usuario decide cerrar una ventana por cualquier medio, pero antes de que se produzca el cierre. En el manejador de este evento podemos evitar el cierre de la ventana mediante la asignación `e.Cancel = True`. Si el cierre no se cancela, se producirá a continuación el evento *Closed*.

También cabe destacar el evento *Activated*, que se produce cuando la ventana pasa a ser ventana activa para el usuario.

1. Ventanas

Aplicaciones multiventana

```
private void AbrirButton_Click(object sender, RoutedEventArgs e)
{
    //Creamos una nueva ventana hija
    ChildWindow hija = new ChildWindow();

    //Establecemos la ventana principal como propietaria de la nueva ventana
    hija.Owner = this;

    //Configuramos la ventana hija
    hija.ShowInTaskbar = false;
    hija.WindowStartupLocation = WindowStartupLocation.CenterOwner;

    //Mostramos la nueva ventana
    //Con ShowDialog() se mostraría como modal
    hija.ShowDialog();

    //La colección OwnedWindows permite acceder a las ventanas hijas
    AbiertasTextBlock.Text = this.OwnedWindows.Count.ToString();
}
```

Hasta ahora hemos desarrollado aplicaciones con una única ventana, pero en muchas ocasiones necesitaremos aplicaciones con múltiples ventanas.

En primer lugar, deberemos agregar al proyecto una nueva ventana y añadir el XAML y el código trasero necesario. Una vez hecho esto, desde la ventana principal podremos crear una nueva instancia de la ventana hija cuando sea necesario, tal y como se muestra en la diapositiva.

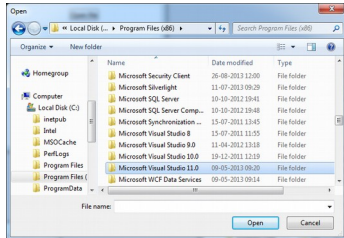
La relación entre las dos ventanas se establece mediante la propiedad *Owner* en la ventana hija. Además, podremos configurar la ventana hija con las propiedades necesarias, antes de mostrarla con el método *Show*.

La ventana padre tiene la propiedad *OwnedWindows* mediante la que acceder a todas sus ventanas hijas.

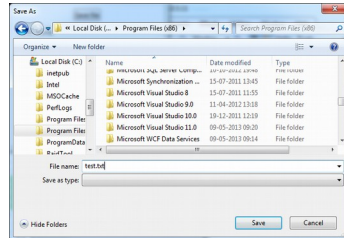
2. Diálogos

Diálogos predefinidos

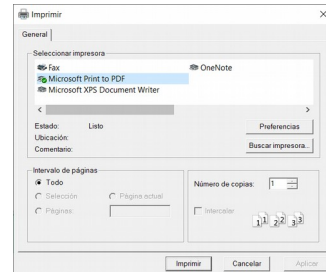
OpenFileDialog



SaveFileDialog



PrintDialog



Existe un tipo especial de ventana denominado diálogo, que se utilizan para realizar una acción concreta con el usuario.

WPF incorpora tres diálogos predefinidos para acciones habituales en la mayoría de las aplicaciones:

- *OpenFileDialog*: para seleccionar un archivo para abrir.
- *SaveFileDialog*: para seleccionar donde guardar un archivo.
- *PrintDialog*: para lanzar una tarea de impresión a la impresora.

2. Diálogos

Diálogos predefinidos - OpenFileDialog

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string rutaFichero;

    //Creamos el nuevo diálogo
    Microsoft.Win32.OpenFileDialog dialogo = new Microsoft.Win32.OpenFileDialog();

    //Configuramos el filtro del diálogo
    dialogo.Filter = "Archivos de texto|.txt|Todos los archivos|*.*";
    //Configuramos el directorio inicial
    dialogo.InitialDirectory = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

    //Mostramos el diálogo
    if (dialogo.ShowDialog() == true)
        //Si el usuario ha pulsado en Aceptar, tenemos la ruta del fichero en FileName
        rutaFichero = dialogo.FileName;
}
```

En este ejemplo se muestra el uso del *OpenFileDialog*. Como se puede observar, es necesario crear una nueva instancia del diálogo. Antes de mostrarlo, podemos configurarlo con diversas propiedades, como *Filter* (para establecer un filtro al tipo de ficheros) o *InitialDirectory* (para determinar el directorio inicial que se mostrará).

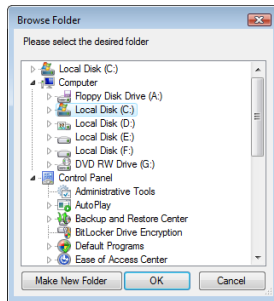
Una vez configurado lo podemos mostrar al usuario con el método *ShowDialog*. Este método mostrará el diálogo como ventana modal, lo que implica que el usuario no podrá utilizar el resto de nuestra aplicación hasta que termine la acción del diálogo.

El método *ShowDialog* devuelve un booleano indicando si el usuario ha aceptado o cancelado el diálogo. Una vez cancelado, en la propiedad *FileName* estará la ruta completa seleccionada por el usuario.

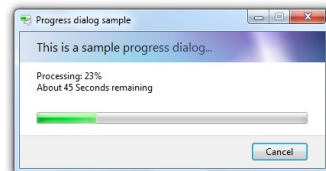
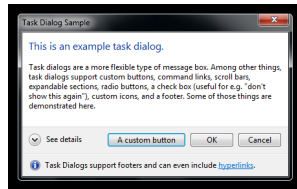
2. Diálogos

Diálogos predefinidos – Otros diálogos

FolderBrowserDialog (WinForms)



Ookii Dialogs (NuGet)



Además de las tres tareas comentadas, existen otras tareas que normalmente se implementan mediante diálogos para las que no existe un diálogo predefinido en WPF. No obstante, podemos recurrir a otras opciones para evitar tener que implementar esos diálogos nosotros mismos.

Este es el caso del diálogo de selección de directorio que ya hemos utilizado, que pertenece a la tecnología *WinForms*.

Para otro tipo de tareas podemos recurrir a desarrollos de terceros (normalmente ofrecidos como paquetes *NuGet*) como los *Ookii Dialogs*, que incluyen diálogos para tareas, progresos o introducción de credenciales.

2. Diálogos

Diálogos personalizados

Se crean como ventanas corrientes, pero se configuran para aparecer como diálogos:

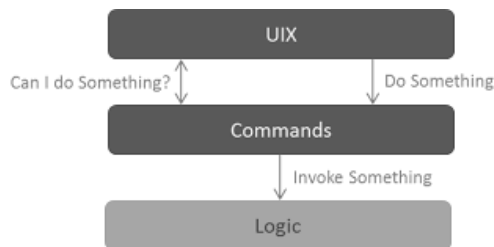
- Se muestran con *ShowDialog*
- Se debe configurar el control que debe tener el foco al mostrarlo
- No aparecen en la barra de tareas
- Configurar los botones por defecto (propiedades *IsDefault* and *IsCancel*)
- En el manejador del botón *Aceptar* establecer *DialogResult* a *true*
- En la mayoría de ocasiones, deben ser de tamaño fijo (*ResizeMode=NoResize*)

En muchas ocasiones, nuestra aplicación necesitará diálogos con una funcionalidad personalizada que no esté cubierta por ningún diálogo predefinido o existente en NuGet. En estos casos, deberemos desarrollar nosotros mismos el diálogo.

Al fin y al cabo, un diálogo no es más que una ventana de WPF, configurada adecuadamente para que su aspecto y comportamiento visual sea consistente con el de los diálogos habituales.

En la diapositiva se muestran algunos de los aspectos que permitirán a la ventana comportarse como un diálogo. Importante destacar que debemos devolver el resultado del diálogo cuando el usuario presione *Aceptar*. Al cancelar no es necesario si se ha establecido *IsCancel* en el botón.

3. Comandos



- Permiten asociar una misma acción a distintos gestos de la IU (opción de menú, botón, atajo de teclado,...).
- Evitan duplicar código en diferentes manejadores de eventos.
- Permiten deshabilitar automáticamente la interfaz cuando el comando no está disponible.

Los comandos son un mecanismo ofrecido por WPF para asociar una misma acción a diferentes gestos que el usuario pueda hacer en la interfaz (como una opción de menú, un botón de una barra de herramientas o un atajo de teclado).

El objetivo de los comandos es separar la semántica de una acción de la forma de invocar la acción. De esta forma, evitaremos duplicar código en diferentes partes de la aplicación.

Otra funcionalidad añadida que ofrecen los comandos es la de indicar si la acción está disponible. Cuando no lo está, todos los elementos de la interfaz asociados al comando cambian automáticamente su estado para reflejarlo.

3. Comandos

Comandos predefinidos

Class	Command Types
ApplicationCommands	Commands common to almost all applications. Includes Clipboard commands, undo, redo and document level operations (open, close, print, etc...).
ComponentCommands	Operations for moving through information, such as scroll up and down, move to end, and text selection.
EditingCommands	Text Editing Commands such as: bold, italics, center and justify.
MediaCommands	Media Playing operations such as (play, pause, etc...), volume control and track selection.
NavigationCommands	Browser-like navigation commands such as Back, Forward and Refresh.

Existen en WPF 5 categorías de comandos predefinidos que podemos utilizar en nuestras aplicaciones:

- *ApplicationCommands*: comandos generales como manejo del portapapeles (cortar, copiar y pegar), deshacer y rehacer, ...
- *ComponentCommands*: comandos para moverse por información, como avanzar o retroceder página, ir al final y selección de texto.
- *EditingCommands*: opciones de edición de texto como negrita, cursiva, centrado o justificado.
- *MediaCommands*: acciones relacionadas con reproducción de multimedia como iniciar, pausar o control del volumen.
- *NavigationCommands*: comandos de navegación como atrás, adelante y refrescar.

3. Comandos

Comandos predefinidos – Comportamiento por defecto

```
<StackPanel>
  <TextBox x:Name="EditorTextBox"></TextBox>
  <Button x:Name="CopiarButton"
    Command="ApplicationCommands.Copy"
    CommandTarget="{Binding ElementName=EditorTextBox}">
    Copiar
  </Button>
  <Button x:Name="PegarButton"
    Command="ApplicationCommands.Paste"
    CommandTarget="{Binding ElementName=EditorTextBox}">
    Pegar
  </Button>
</StackPanel>
```

Los comandos predefinidos se puede utilizar de dos formas. Una de ellas es utilizando el comportamiento por defecto que cada comando tiene asociado.

Para ello, asociaremos el comando al control que debe ejecutarlo (por ejemplo, un botón) mediante la propiedad *Command*. Mediante la propiedad *CommandTarget* indicaremos a qué control afectará la acción del comando. Dependiendo del comando, el *CommandTarget* podrá ser un tipo de control u otro.

En el ejemplo, tenemos dos botones, cada uno de ellos con un comando de edición diferente (copiar y pegar) ambos asociados a un mismo cuadro de texto.

3. Comandos

Comandos predefinidos – Comportamiento personalizado

```
<Window.CommandBindings>
  <CommandBinding
    Command="ApplicationCommands.New"
    Executed="NewCommand_Executed"
    CanExecute="NewCommand_CanExecute" />
</Window.CommandBindings>

<StackPanel>
  <StackPanel x:Name="ContenedorStackPanel"></StackPanel>
  <Button x:Name="AñadirButton" Command="ApplicationCommands.New">Añadir etiqueta</Button>
</StackPanel>
```

Otra posibilidad cuando utilizamos los comandos predefinidos es la de implementar nosotros el comportamiento del comando. Para ello, tendremos que crear un nuevo *CommandBinding* en la sección *CommandBindings* de la ventana.

En el *CommandBinding* se define el método que llevará a cabo la acción del comando (propiedad *Executed*) y el método que decidirá si el comando está disponible o no (*CanExecute*).

Posteriormente, podremos asociar un control de la interfaz al comando definido.

3. Comandos

Comandos predefinidos – Comportamiento personalizado

```
private void NewCommand_Executed(object sender, ExecutedRoutedEventArgs e)
{
    TextBlock etiqueta = new TextBlock();
    etiqueta.Text = "Etiqueta " + (ContenedorStackPanel.Children.Count + 1).ToString();
    ContenedorStackPanel.Children.Add(etiqueta);
}

private void NewCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    if (ContenedorStackPanel.Children.Count < 10)
        e.CanExecute = true;
    else
        e.CanExecute = false;
}
```

En el código trasero de la ventana tendremos la implementación de los dos métodos definidos en el comando.

Por un lado, el método que llevará a cabo la acción del comando (en el ejemplo *NewCommand_Executed*). En este caso, simplemente crea un nuevo *TextBlock* y lo añade a un *StackPanel*.

Por otro lado, tenemos la implementación del método que decidirá si el comando está disponible (*NewCommand_CanExecute* en este caso). En el ejemplo, el comando no debe estar disponible si ya hay diez etiquetas en el contenedor. Para indicar la disponibilidad del comando se utiliza la propiedad booleana *e.CanExcute*.

3. Comandos

Comandos personalizados

```
public static class CustomCommands
{
    public static readonly RoutedUICommand Delete = new RoutedUICommand
    (
        "Delete", // Etiqueta
        "Delete", // Nombre
        typeof(CustomCommands), // Clase contenedora
        new InputGestureCollection()
        {
            new KeyGesture(Key.D, ModifierKeys.Control) // Atajo
        }
    );
}
```

Cuando ninguno de los comandos predefinidos es adecuado para una acción de nuestra aplicación, tenemos la posibilidad de crear un comando personalizado.

Para ello, lo más habitual es utilizar una clase estática contenedora (en el ejemplo la clase *CustomCommands*) dentro de la cual definiremos cada comando como un campo estático de tipo *RoutedUICommand*. En el constructor del comando indicaremos la siguiente información:

- Etiqueta del comando (*Delete*)
- Nombre del comando (*Delete*)
- Clase contenedora (*CustomCommands*)
- Atajo de teclado (Ctrl+D)

3. Comandos

Comandos personalizados

```
<Window.CommandBindings>
  <CommandBinding
    Command="ApplicationCommands.New"
    Executed="NewCommand_Executed"
    CanExecute="NewCommand_CanExecute" />
  <CommandBinding
    Command="local:CustomCommands.Delete"
    Executed="DeleteCommand_Executed"
    CanExecute="DeleteCommand_CanExecute" />
</Window.CommandBindings>

<StackPanel>
  <StackPanel x:Name="ContenedorStackPanel"></StackPanel>
  <Button x:Name="AñadirButton" Command="ApplicationCommands.New">Añadir etiqueta</Button>
  <Button x:Name="QuitarButton" Command="local:CustomCommands.Delete">Quitar etiqueta</Button>
</StackPanel>
```

Una vez definido el comando ya podremos utilizarlo como si fuer un comando predefinido. Definiremos el *CommandBinding* correspondiente, y lo asociaremos mediante la propiedad *Command* a uno o varios controles de nuestra interfaz.

3. Comandos

Comandos personalizados

```
private void DeleteCommand_Executed(object sender, ExecutedRoutedEventArgs e)
{
    ContenedorStackPanel.Children.RemoveAt(ContenedorStackPanel.Children.Count - 1);
}

private void DeleteCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    if (ContenedorStackPanel.Children.Count > 0)
        e.CanExecute = true;
    else
        e.CanExecute = false;
}
```

Una vez definido el *CommandBinding* en XAML solo quedará implementar en el código trasero los métodos definidos.

En este caso, el método que lleva a cabo la lógica del comando es *DeleteCommand_Executed*, que eliminará la última etiqueta añadida al contenedor.

El método que decide si el comando está disponible es *DeleteCommand_CanExecute*, que en este caso comprueba si hay alguna etiqueta en el contenedor.

4. Menús

```
<DockPanel>
  <Menu DockPanel.Dock="Top">
    <MenuItem Header="_Archivo">
      <MenuItem Header="_Nuevo">
        <MenuItem.Icon>
          <TextBlock FontFamily="Wingdings">2</TextBlock>
        </MenuItem.Icon>
      </MenuItem>
      <MenuItem Header="_Abrir..." InputGestureText="Ctrl+A"/>
      <MenuItem Header="_Corrector" IsCheckable="True" IsChecked="True"/>
      <Separator />
      <MenuItem Header="_Salir..." Click="SalirMenuItem_Click"/>
    </MenuItem>
    <MenuItem Header="_Editar">
      <MenuItem Header="_Copiar" Command="ApplicationCommands.Copy" />
      <MenuItem Header="_Cortar" Command="ApplicationCommands.Cut"/>
      <MenuItem Header="_Pegar" Command="ApplicationCommands.Paste"/>
    </MenuItem>
  </Menu>
  <Grid>
    <!-- Contenido de la ventana -->
  </Grid>
</DockPanel>
```

Solo a nivel informativo.

Para que funcione:

-Comando

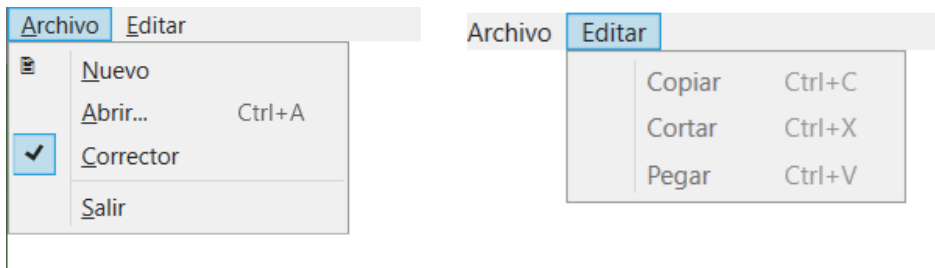
o

- Capturar evento teclado

Crear un menú en WPF es relativamente sencillo. Los menús se definen con la etiqueta *Menu*, que contendrá elementos *MenuItem* para definir su contenido. Los elementos *MenuItem* se pueden anidar para conseguir la estructura de menús y submenús deseada.

Existen diferentes propiedades del *MenuItem* que nos permiten configurar el menú. La propiedad **Header** establece la etiqueta que aparece en la opción de menú. Si se antepone un guión bajo (_) a alguna de las letras, ésta será utilizada para el acceso rápido a dicha opción. También cabe destacar que si la opción de menú implica la apertura de un diálogo para llevar a cabo su acción la etiqueta debe acabar con puntos suspensivos ('...').

4. Menús



Otra propiedad del MenuItem es **Icon**, que nos permite asociar un icono en la parte izquierda del item. Como se puede apreciar en el ejemplo, el valor de esta propiedad puede ser cualquier control (incluso un contenedor).

Con la propiedad **IsCheckable** podemos conseguir que el item sea seleccionable, de forma similar a un *CheckBox*. La casilla de verificación aparecerá en la parte izquierda.

La propiedad **InputGestureTest** asocia un atajo de teclado a la opción de menú, pero solo a nivel informativo (aparecerá a la derecha del item). Para que realmente funcione podríamos asociar un comando al item con dicho atajo (opción recomendada) o capturar el evento de pulsación de teclas y comprobar si se da dicha combinación.

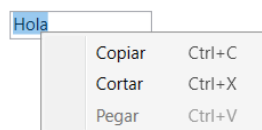
Para asignar la funcionalidad al item podemos optar por asociarle un comando (opción recomendada) o controlar el evento Click del item.

4. Menús

Menú contextual

```
<Window.Resources>
    <ContextMenu x:Key="EdiciónContextMenu">
        <MenuItem Header="Copiar" Command="ApplicationCommands.Copy"/>
        <MenuItem Header="Cortar" Command="ApplicationCommands.Cut"/>
        <MenuItem Header="Pegar" Command="ApplicationCommands.Paste"/>
    </ContextMenu>
</Window.Resources>

<Grid>
    <TextBox ContextMenu="{StaticResource EdiciónContextMenu}"></TextBox>
</Grid>
```



Los menús contextuales están asociados a un control concreto, y aparecen cuando pulsamos con el botón derecho del ratón sobre el control.

En WPF, estos menús se definen con el elemento *ContextMenu*, que está compuesto de elementos *MenuItem* como los menús normales, con las mismas posibilidades que hemos visto.

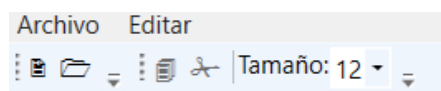
Normalmente se definen en los recursos de la aplicación o la ventana, para así poder ser reutilizados en diferentes controles.

Para asociar un menú a un control se utiliza la propiedad *ContextMenu* del control.

5. Barra de herramientas y barra de estado

Barra de herramientas

```
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar>
    <Button FontFamily="Wingdings" ToolTip="Nuevo">2</Button>
    <Button FontFamily="Wingdings" ToolTip="Abrir">1</Button>
  </ToolBar>
  <ToolBar>
    <Button Command="ApplicationCommands.Copy" FontFamily="Wingdings" ToolTip="Copiar">4</Button>
    <Button Command="ApplicationCommands.Cut" FontFamily="Wingdings" ToolTip="Pegar">#</Button>
    <Separator />
    <TextBlock>Tamaño:</TextBlock>
    <ComboBox>
      <ComboBoxItem>10</ComboBoxItem>
      <ComboBoxItem>12</ComboBoxItem>
    </ComboBox>
  </ToolBar>
</ToolBarTray>
```



Además de los menús, es muy habitual que las aplicaciones cuenten con barras de herramientas para que los usuarios puedan acceder a acciones comunes rápidamente.

Las barras de herramientas se agrupan en un control llamado *ToolBarTray*, que dentro contendrá elementos *ToolBar*. Dentro de la barra, podemos tener controles de cualquier tipo, aunque lo más habitual serán botones con una imagen como contenido. Además podemos insertar un separador entre dos componentes de la barra con el elemento *Separator*.

Como norma general, todas las opciones disponibles en la barra de herramientas deberían ser accesible desde el menú, y la imagen asociada ser consistente en ambos elementos.

5. Barra de herramientas y barra de estado

Barra de estado

```
<StatusBar DockPanel.Dock="Bottom">
  <StatusBarItem>
    <TextBlock Text="Diseñado por: Javier Catalá"></TextBlock>
  </StatusBarItem>
  <Separator />
  <StatusBarItem>
    <StackPanel Orientation="Horizontal">
      <TextBlock>Guardando...</TextBlock>
      <ProgressBar Width="200" Value="50"></ProgressBar>
    </StackPanel>
  </StatusBarItem>
</StatusBar>
```

Diseñado por: Javier Catalá

Guardando...



También es común que las aplicaciones contengan una barra de estado. Normalmente se ubica en la parte inferior de la ventana, y suele incluir información de la aplicación y algún tipo de funcionalidad básica.

Para crearla se utiliza el elemento *StatusBar*, que dentro contiene elementos *StausBarItem*. Cada item puede contener cualquier control (incluso un contenedor). Y también tenemos disponible el elemento *Separator*.

6. Configuración de aplicaciones

La configuración de la aplicación permite almacenar y recuperar valores de configuración. Por ejemplo, la aplicación puede guardar las preferencias de los usuarios. [Más acerca de la configuración de la aplicación...](#)

Nombre	Tipo	Ámbito	Valor
servidor	string	Aplicación	192.168.0.1
color	string	Usuario	White
*			

- Valor común para todos los usuarios
- No se puede modificar durante la ejecución

- Valor diferente para cada usuario
- Se puede modificar durante la ejecución

En la mayoría de aplicaciones surge la necesidad de almacenar parámetros de configuración, que no conviene tener directamente en el código. Algunos ejemplos típicos son cadenas de conexión a base de datos o rutas de directorios de log.

Para crear variables de configuración, nos dirigiremos a la sección Configuración de las propiedades del proyecto en Visual Studio. Podremos crear configuraciones de dos ámbitos:

- **Aplicación:** tendrá el mismo valor para todos los usuarios, y no se puede cambiar en tiempo de ejecución.
- **Usuario:** tendrá un valor distinto para cada usuario, y puede ser modificado en la ejecución.

6. Configuración de aplicaciones

Acceso mediante binding

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApp1"
        xmlns:properties="clr-namespace:WpfApp1.Properties"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <StackPanel Background="{Binding Source={x:Static properties:Settings.Default},Path=color}">
        <TextBlock>
            <Run>Servidor</Run>
            <Run Text="{Binding Source={x:Static properties:Settings.Default},Path=servidor,Mode=OneWay}"></Run>
        </TextBlock>
        <Button x:Name="CambiarButton" Click="CambiarButton_Click">Cambiar color</Button>
    </StackPanel>
</Window>
```

Podemos acceder a las variables de configuración creadas desde XAML (mediante *binding*) o desde el código.

En la diapositiva se muestra un ejemplo de uso desde XAML. Vemos que es necesario añadir un nuevo *namespace* para acceder a las propiedades del proyecto.

Cuando desde un *binding* queramos hacer referencia a la configuración usaremos la propiedad *Source* del *binding* para hacer referencia a la configuración del proyecto, y el *Path* para indicar el nombre dado a la variable de configuración.

Si enlazamos una variable de ámbito Aplicación, y el *binding* es por defecto *TwoWay* (como en segundo caso del ejemplo) deberemos especificar el modo *OneWay*, ya que las variables de Aplicación no se pueden modificar.

6. Configuración de aplicaciones

Acceso desde code behind

```
private void CambiarButton_Click(object sender, RoutedEventArgs e)
{
    Properties.Settings.Default.color = "Blue";
    Properties.Settings.Default.Save();
}
```

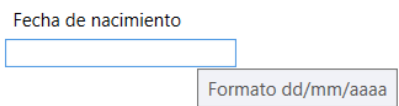
Como se puede apreciar en el ejemplo, el acceso a las variables de configuración desde el *code behind* es muy sencillo.

Tener en cuenta que si se está modificando una variable de tipo Usuario, los cambios no se almacenarán realmente en la configuración del usuario si no se invoca el método *Save*.

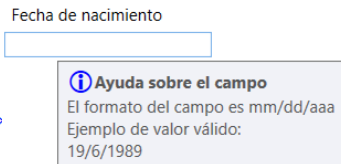
En el proyecto, las variables de configuración se almacenan en el fichero *App.config*. Una vez compilada la aplicación, se crea en la carpeta de salida un fichero con el mismo nombre que el ejecutable y extensión *.config*. Este fichero puede ser modificado para cambiar la configuración, sin necesidad de recompilar.

7. Tooltips

```
<Label>Fecha de nacimiento</Label>
<TextBox x:Name="FechaTextBox" ToolTip="Formato dd/mm/aaaa"
        ToolTipService.ShowOnDisabled="True"
        ToolTipService.ShowDuration="5000" />
```



```
<Label>Fecha de nacimiento</Label>
<TextBox x:Name="FechaTextBox">
    <TextBox.ToolTip>
        <StackPanel>
            <StackPanel Orientation="Horizontal">
                <TextBlock FontFamily="Webdings" Foreground="Blue"
                        FontSize="20" VerticalAlignment="Center">i</TextBlock>
                <TextBlock FontWeight="Bold" VerticalAlignment="Center">
                    Ayuda sobre el campo</TextBlock>
            </StackPanel>
            <TextBlock>El formato del campo es mm/dd/aaa</TextBlock>
            <TextBlock>Ejemplo de valor válido:</TextBlock>
            <TextBlock>19/6/1989</TextBlock>
        </StackPanel>
    </TextBox.ToolTip>
</TextBox>
```



Tema 5. Aplicaciones WPF

26

Los *tooltips* nos permiten añadir información a un control, normalmente para aclarar al usuario su función. Para establecerlos se utiliza la propiedad *ToolTip*, presente en la mayoría de los controles.

El primer ejemplo muestra su uso más simple, en el que el contenido del *tooltip* es una cadena de texto. Como se puede apreciar, el *tooltip* se puede configurar con propiedades adjuntas, por ejemplo para decidir si debe mostrarse en caso de que el control esté deshabilitado o para controlar el tiempo que se muestra.

En el segundo ejemplo podemos ver que realmente el valor de esta propiedad no tiene por qué restringirse a una cadena, y el *tooltip* puede tener la estructura que deseemos. Puede ser útil definir un *UserControl* para que todos nuestros *tooltips* tengan la misma estructura.

8. Mejora de la productividad

Uso del tabulador

```
<StackPanel Width="150" Margin="20">
    <Label>Nombre</Label>
    <TextBox x:Name="NombreTextBox" TabIndex="1"/>
    <Label>Fecha de nacimiento</Label>
    <TextBox x:Name="FechaTextBox" TabIndex="2"/>
    <Button TabIndex="3">Aceptar</Button>
    <Button IsTabStop="False">Cancelar</Button>
</StackPanel>
```

TabIndex permite controlar el orden de tabulación

Con IsTabStop podemos excluir un control del orden de tabulación

Para finalizar el tema vamos a ver dos características que permiten mejorar la productividad de los usuarios, sobre todo en aplicaciones empresariales: el uso del tabulador y el método de acceso rápido a controles.

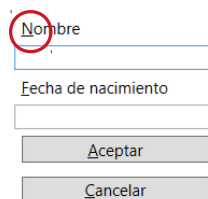
Muchos usuarios están habituados a utilizar el tabulador para moverse por los controles de una ventana. Por ello, es importante que establezcamos un orden lógico al recorrido que se realizará por los controles. Para controlar este recorrido utilizamos la propiedad *TabIndex* de los controles, que nos permitirá establecer el orden deseado.

Con la propiedad *IsTabStop* podemos controlar si un control debe excluirse del orden de tabulación (dándole valor *false*).

8. Mejora de la productividad

Acceso rápido

```
<StackPanel Width="150" Margin="20">
    <Label Target="{Binding ElementName=NombreTextBox}">_Nombre</Label>
    <TextBox x:Name="NombreTextBox"/>
    <Label Target="{Binding ElementName=FechaTextBox}">_Fecha de nacimiento</Label>
    <TextBox x:Name="FechaTextBox"/>
    <Button>_Aceptar</Button>
    <Button>_Cancelar</Button>
</StackPanel>
```



De forma análoga a como vimos en los items de los menús, es posible asociar un carácter de acceso rápido a otros elementos como botones o etiquetas. El funcionamiento es similar, basta con anteponer un guión bajo a la letra que utilizaremos para el acceso rápido.

Como vemos en el ejemplo para las etiquetas, también es posible asociar mediante la propiedad *Target* un destino diferente al control cuando se utiliza el acceso rápido. Para ello utilizamos una expresión de *binding* para referenciar al control asociado.

Para utilizar esta funcionalidad, como también ocurre en los menús, es necesario pulsar la tecla *Alt*.